**Name : Omkar Nikhal**

**Class : SE-3**

**Roll no : 21342**

**ASSIGNMENT NO.6**

|  |  |
| --- | --- |
| Title: | The Dictionary ADT |
| Problem Statement: | Implement all the functions of a dictionary (ADT) using hashing. Data: Set of (key, value) pairs, Keys are mapped to values, Keys must be comparable, Keys must be unique Standard Operations: Insert (key, value), Find(key), Delete(key) |
| Objective: | To understand implementation of all the functions of a dictionary (ADT) and standard operations on Dictionary. |
| Outcome: | At the end of this assignment students will able to perform standard operations on Dictionary ADT. |
| S/W Packages and H/W Apparatus used: | 1. 64-bit Fedora 17 or latest 64-Bit update of equivalent open source OS 2. Programming tools (64-Bit) and latest open source update of Eclipse Programming framework, TC++, GTK++ |

Theory

The Dictionary ADT:

A dictionary is an ordered or unordered list of key-element pairs, where keys are used to locate elements in the list.

Dictionary is a data structure, which is generally an association of unique keys with some values. One may bind a value to a key, delete a key (and naturally an associated value) and look up for a value by the key. Values are not required to be unique.

Example: consider a data structure that stores bank accounts; it can be viewed as a dictionary, where account numbers serve as keys for identification of account objects.

A Dictionary (also known as Table or Map) can be implemented in various ways: using a list, binary search tree, hash table, etc.

In each case: the implementing data structure has to be able to hold key-data pairs and able to do insert, find, and delete operations paying attention to the key.

Hashing

Hashing is a method for directly referencing an element in a table by performing arithmetic transformations on keys into table addresses. This is carried out in two steps:

1. Computing the so-called hash function H: K -> A.
2. Collision resolution, which handles cases where two or more different keys hash to the same table address.

Implementation of Hash Table

Hash tables consist of two components: a bucket array and a hash function.

A hash table is a collection of items which are stored in such a way as to make it easy to find them later. Each position of the hash table, often called a slot, can hold an item and is named by an integer value starting at 0. For example, we will have a slot named 0, a slot named 1, a slot named 2, and so on.

Consider a dictionary, where keys are integers in the range [0, N-1]. Then, an array of size N can be used to represent the dictionary. Each entry in this array is thought of as a “bucket”. An element e with key k is inserted in A[k]. Bucket entries associated with keys not present in the dictionary contain a special NO\_SUCH\_KEY object. If the dictionary contains elements with the same key, then two or more different elements may be mapped to the same bucket of A. In this case, we say that a collision between these elements has occurred. One easy way to deal with collisions is to allow a sequence of elements with the same key, k, to be stored in A[k].

Assuming that an arbitrary element with key k satisfies queries findItem(k) and removeItem(k), these operations are now performed in O(1) time, while insertItem(k, e) needs only to find where on the existing list A[k] to insert the new item, e. The drawback of this is that the size of the bucket array is the size of the set from which key are drawn,which may be huge.

Algorithms

HashNode Class Declaration

class HashNode{

public:

int key;

int value;

HashNode \* next;

HashNode(int key, int value){

this->key = key;

this->value = value;

this->next = NULL;

}

};

Insertion

void Insert(int key, int value){

int hash\_val = HashFunc(key);

HashNode \* prev = NULL;

HashNode \* entry = htable[hash\_val];

while(entry != NULL){

prev = entry;

entry = entry->next;

}

if(entry == NULL){

entry = new HashNode(key,value);

if(prev == NULL){

htable[hash\_val] = entry;

}

else{

prev->next = entry;

}

}

else{

entry->value = value;

}

}

Deletion

void Remove(int key){

int hash\_val = HashFunc(key);

HashNode\* entry = htable[hash\_val];

HashNode\* prev = NULL;

if (entry == NULL || entry->key != key){

cout<<"No Element found at key "<<key<<endl;

return;

}

while (entry->next != NULL){

prev = entry;

entry = entry->next;

}

if (prev != NULL){

prev->next = entry->next;

}

delete entry;

cout<<"Element Deleted"<<endl;

}

Search

int Search(int key){

bool flag = false;

int hash\_val = HashFunc(key);

HashNode\* entry = htable[hash\_val];

while (entry != NULL){

if (entry->key == key){

cout<<entry->value<<" ";

flag = true;

}

entry = entry->next;

}

if (!flag)

return -1;

}

Test-Cases

|  |  |  |  |
| --- | --- | --- | --- |
| Description | Input | Output | Result |
| Insert without replacement | Key: 0, Value: Apple  Key: 1, Value: Banana  Key: 27, Value: Berry  Key: 2, Value: Cherry | |  |  |  |  | | --- | --- | --- | --- | |  | Key | Hash %26 | Value | | 0 | 0 | 0 | Apple | | 1 | 1 | 1 | Banana | | 2 | 27 | 1 | Berry | | 3 | 2 | 2 | Cherry | | Pass |
| Insert with replacement | Key: 0, Value: Apple  Key: 1, Value: Banana  Key: 27, Value: Berry  Key: 2, Value: Cherry | |  |  |  |  | | --- | --- | --- | --- | |  | Key | Hash %26 | Value | | 0 | 0 | 0 | Apple | | 1 | 1 | 1 | Banana | | 2 | 2 | 2 | Cherry | | 3 | 27 | 1 | Berry | | Pass |
| Search | Enter key: 27  Enter key: 1 | Element “Berry” found at key: 27  Element “Banana” found at key: 1 | Pass |
| Delete | Enter key: 2  Enter key: 0 | Element “Cherry” deleted from table  Element “Apple” deleted from table | Pass |

Conclusion

After successfully completing this assignment, students have learned implementation of Dictionary (ADT) using Hashing and various standard operations on Dictionary ADT